the claimed terminating including "removing the first data from the first data structure", where the first data ***identifies information based on execution of the instance***."

Regarding the second feature of "removing the first data from the first data structure", Applicant also traverses the Official Action as incomplete because it fails to answer the material traversed. (See MPEP §707.07(f) "Where the applicant traverses any rejection, the examiner should, if he or she repeats the rejection, take note of the applicant's argument and answer the substance of it."). In particular, the Final Action on page 13 notes Applicant's argument that the Examiner disregards "removing the first data from the first data structure", but *fails to address this argument*, rather focusing on Applicant's argument regarding the modification of Gibson (discussed below).

Applicant specifically rebutted the Examiner's assertion that the claimed "removing the first data from the first data structure" was disclosed at col. 4, lines 59-64 of Gibson (see, e.g., page 3, line 15 of Final Action), by demonstrating in specific detail on pages 13-15 of Applicant's January 18, 2006 response not only that the cited col. 4, lines 59-64 described nothing more than detecting the type of incoming call[1], but also why Gibson consistently maintained storage of ***all relevant data***, with no removal whatsoever. As noted by the Examiner on page 13:

> Gibson shows the storage/***retention*** of misdialed CLIs and transmission of a voice message regardless of detected call type are performed by Gibson to facilitate further processing of misdialed calls.

Although the storage/retention of misdialed CLIs is not relevant "to the basic operations of call-type detection and processing discussed in Applicant's Specification" (and therefore not a teaching of the claimed features), such storage/retention ***is relevant to establishing that Gibson does not teach or suggest the claimed "removing the first data from the first data structure."*** [2] Contrary

---

[1]Col. 4, lines 59-64 recite *in toto*: "After the CLI capture facility 350, the call 300 is then monitored by a fax detection facility 305 which operates to detect whether the call 300 is a voice call, a fax call or a data call. The fax detection facility 305 then advises the message selector 310 whether or not the incoming call 300 is a fax call or a voice call."

[2]"A prior art reference must be considered in its entirety, i.e., as a whole, including portions that would lead away from the claimed invention. MPEP §2141.02, page 2100-132 (Rev. 3, Aug. 2005) (*citing W.L. Gore & Assoc. v. Garlock, Inc.*, 220 USPQ 303 (Fed. Cir.

to the Examiner's assertions, the CLI data (the "first data" written into the first data structure) is never shown to be removed as part of the terminating step, but in fact is ***stored for subsequent analysis*** (see, e.g., Fig. 5 and col. 9, lines 44-52).

Hence, the Final Action is legally deficient because it fails to identify how Gibson teaches or suggests "removing the first data from the first data structure" as specified in claims.

As admitted in the Final Action on page 4, "Gibson does not explicitly show selectively terminating the instance prior to completing the sequence of messaging operations." The Final Action also states this admission after acknowledging "the incoming call constitutes an initiating of messaging for ***both voice and fax call types***" and that "[i]n Fig. 4a-b Gibson shows that a voice message is downloaded and transmitted to the misdialing terminal *regardless of whether the call is determined to be a fax call or a voice call.*"

This acknowledgement **unequivocally demonstrates** that the Examiner admits that Gibson teaches initiating a messaging session **that is uses for both voice and fax call types**, and that the messaging session at step 475 of Fig. 4b **either continues** to execute steps 485-4970 for fax calls, or **normally terminates** in step 480 if the call is not a fax call.

The Examiner also demonstrates an unreasonable interpretation of the claimed "instance" as a simple execution step, instead of the broadest ***reasonable interpretation*** and explicitly claimed that the application instance executes the prescribed sequence of messaging operations:

> Gibson discloses that the sequence of operations for subsequent call processing is ***predicated on the result of the Fax detection facility 305***, such that ***the standard "instance"*** [sic] ***for a fax call either continues or terminates***.

This statement demonstrates the Examiner's mischaracterization of Gibson and the claims: the sequence of operations to be performed depends on the result of the Fax detection facility is a

---

1983), *cert. denied*, 469 U.S. 851 (1984))(emphasis in original).

Although the test for establishing an implicit motivation in the prior art is what a prior art statement would have suggested to those of ordinary skill, such a statement "must be considered ***in the context of the teaching of the entire reference***." *In re Kotzab*, 55 USPQ2d 1313, 1317 (Fed. Cir. 2000).

simple logical flow of a single, persistent application instance that performs different ***logical steps*** depending on whether a fax is detected.

The explicit claim language however, specify that the instance executes ***the prescribed sequence of messaging operations for the first type of incoming message***, and that the instance is terminated ***prior to completing the sequence of messaging operations***. Further, termination of the instance includes removal of the first data that was stored during execution of the instance. Hence, the Examiner's interpretation of "instance" as a *conditional sequence of steps* is inconsistent with the explicit claim language, inconsistent with the specification (e.g., at lines 2, 6, 8 of Abstract), and inconsistent with the interpretation those skilled in the art would reach[3], and therefore unreasonable.[4]

Finally, the rejection fails to demonstrate that one skilled in the art would have been motivated to modify Gibson in order to provide the modification in the manner claimed. The Examiner is reminded that an obviousness rejection requires a specific showing as to why one of ordinary skill in the art would have selected the components for combination in the manner claimed. "The mere fact that the prior art may be modified in the manner suggested by the Examiner does not make the modification obvious unless the prior art suggested the desirability of the modification." *In re Fritch*, 23 USPQ2d 1780, 1783-84 (Fed. Cir. 1992). *In re Mills*, 16 USPQ2d 1430 (Fed. Cir. 1990).

Gibson performs the following operations:

(1) determines whether a transmitting station (e.g., fax terminal 100 of Fig. 1) has caused a

---

[3]An "instance" of a program is recognized as a program running on a computer, see attached Exhibit A from http://en.wikipedia.org/wiki/Instance

[4]"During patent examination, the pending claims must be 'given their broadest reasonable interpretation consistent with the specification.'" MPEP §2111 at 2100-46 (Rev. 3, Aug. 2005) (*quoting In re Hyatt*, 211 F.3d 1367, 1372, 54 USPQ2d 1664, 1667 (Fed. Cir. 2000)).

"The broadest reasonable interpretation of the claims must also be consistent with the interpretation that those skilled in the art would reach." MPEP §2111.01 at 2100-47 (Rev. 3, Aug. 2005) (*citing In re Cortright*, 165 F.3d 1353, 1359, 49 USPQ2d 1464, 1468 (Fed. Cir. 1999)).

dialing error (call monitoring facility 200 of Fig. 2 determines if call is to an invalid destination, col. 4, lines 33-45) (step 405 of Fig. 4a) – if there is no dialing error, then call routing is performed per normal procedures (step 415 of Fig. 4a).

If a dialing error ***is*** detected, Gibson performs the following subsequent steps:

(2) identifies the transmitting station having caused the dialing error (GIRAFF unit 220 of Figs. 2 and 3 captures calling number of terminal 100 and called number, col. 4, lines 39-58, steps 420 and 425 of Fig. 4a),

(3) plays a voice message during the call, ***regardless of whether the call is for a fax call or a voice call*** (steps 460 and 465 of Fig. 4a, col. 7, lines 39-53, col. 9, lines 15-36); and concurrently detects whether the call is a fax call (step 475 of Fig. 4b, col. 9, lines 20-36), and

(4) disconnects the calling fax terminal ***only after having completed playing the voice message*** (col. 5, lines 1-6, step 470 of Fig. 4a, 4950 of Fig. 4b, col. 7, lines 45-53, col. 9, lines 31-36).

(5) ***After the original fax call has been disconnected following completion of playing the voice message*** (step 4955 of Fig. 4b, col. 9, lines 31-43), Gibson starts a <u>new</u> call to send back to the transmitting station a fax message notifying the original transmitting station of the dialing error, based on the determined call being a fax call (steps 4960, 4965, 4970 of Fig. 4b, col. 5, lines 10-17, col. 9, lines 40-43).

As apparent from the foregoing, Gibson requires that the <u>entire voice message</u> is played to the calling fax terminal, ***regardless of whether the call is for a fax call or a voice call.*** Hence, Gibson requires the continued waste of text-to-speech system resources (*cf.* col. 7, lines 40-52), and neither discloses nor suggests the claimed selective termination of an instance <u>prior to completing the sequence of messaging operations</u>, as claimed. Rather, Gibson discloses that the ***entire voice message operation*** is to be performed prior to disconnect.

Hence, the Examiner's specious argument of eliminating "unnecessary call processing" begs the question why one skilled in the art would want to terminate the instance *prior to completing the prescribed sequence* of messaging operations? It is more likely that one skilled in the art would **<u>change the prescribed sequence of messaging operations</u>** to ***avoid*** the downloading of the voice

message, in order to <u>prevent the wasted resources of unnecessarily downloading the voice message that would ***not be used***.</u> Moreover, it is notoriously well known in the art that logical flows (such as illustrated in Gibson) should be <u>completed</u> to avoid transitional or incomplete logical states within executable devices.

As admitted by the Examiner, Gibson relies on a single generic configured for receiving <u>**both**</u> fax and voice calls, and logically chooses subsequent steps for facsimile operations based on whether the call is a voice call or a fax call. Each of the independent claims, however, enable ***multiple instances to be executed concurrently for respective call types,*** improving response times for a possible ***voice call to leave a message*** with the application server. An evaluation of obviousness must be undertaken from the perspective of one of ordinary skill in the art addressing the same problems addressed by the applicant in arriving at the claimed invention. *Bausch & Lomb, Inc. v. Barnes-Hind/Hydrocurve*, 23 USPQ 416, 420 (Fed. Cir. 1986), *cert. denied*, 484 US 823 (1987). Thus, the claimed structures and methods cannot be divorced from the problems addressed by the inventor and the benefits resulting from the claimed invention. *In re Newell*, 13 USPQ2d 1248, 1250 (Fed. Cir. 1989).

For these and other reasons, the §103 rejection should be withdrawn.

It is believed the remaining dependent claims are allowable in view of the foregoing.

In view of the above, it is believed this application is in condition for allowance, and such a Notice is respectfully solicited.

To the extent necessary, Applicant petitions for an extension of time under 37 C.F.R. 1.136. Please charge any shortage in fees due in connection with the filing of this paper, including any missing or insufficient fees under 37 C.F.R. 1.17(a), to Deposit Account No. 50-1130, under Order No. 95-462, and please credit any excess fees to such deposit account.

Respectfully submitted,

Leon R. Turkevich
Registration No. 34,035

Customer No. 23164
(202) 261-1059
Date: Monday, April 3, 2006
(April 2, 2006 = Sunday)

Object (computer science)

From Wikipedia, the free encyclopedia
(Redirected from Instance)

In strictly mathematical branches of computer science the term **object** is used in a purely mathematical sense to refer to any "thing". While this interpretation is useful in the discussion of abstract theory, it is not concrete enough to serve as a primitive datatype in the discussion of more concrete branches (such as programming) that are closer to actual computation and information processing. There, objects are still conceptual entities, but generally correspond directly to a contiguous block of computer memory of a specific size at a specific location. This is because computation and information processing ultimately require a form of computer memory. Objects in this sense are fundamental primitives needed to accurately define concepts such as references, variables, and name binding. This is why the rest of this article will focus on the concrete interpretation of *object* rather than the abstract one.

Note that although a block of computer memory can appear contiguous on one level of abstraction and incontiguous on another, the important thing is that it appears contiguous to the program that treats it as an object. That is, as far as the program is concerned the object must be free of internal references, because otherwise it is no longer a primitive. In other words, object's private storage details must not be exposed to clients of the object, and must be able to change without changes to client code.

Objects exist only within contexts that are aware of them; a piece of computer memory only holds an object if a program treats it as such (for example by reserving it for exclusive use by specific procedures and/or associating a data type with it). Thus, the lifetime of an object is the time during which it is treated as an object. This is why they are still conceptual entities, despite their physical presence in computer memory.

In other words, abstract concepts that do not occupy memory space at runtime are, according to the definition, not objects; e.g., design patterns exhibited by a set of classes, data types in statically typed programs.

To emphasize that an object actually contains meaningful data, a term *data object* is sometimes used to refer to such an object.

Objects in Object-Oriented Programming

In Object-Oriented Programming (OOP), an instance of a program (i.e. a program running in a computer) is treated as a dynamic set of interacting objects. Objects in OOP extend the more general notion of objects described above to include a very specific kind of typing, which among other things allows for:

1. data members that represent the data associated with the object.
2. methods that access the data members in predefined ways.

In the case of most objects, one can access the data members only through the method members, making it easy to guarantee that the data will always remain in a well-defined state (class invariants will be enforced). Some languages do not make distinctions between data members and methods.

In a language where each object is created from a class, an object is called an **instance** of that class. If each object has a type, two objects with the same class would have the same datatype. Creating an instance of a class is sometimes referred to as **instantiating** the class.

A real-world example of an object would be "my dog", which is an instance of a type (a class) called "dog", which

Exhibit A to April 3, 2006 Response After Final
Appln. No. 09/800,476

is a subclass of a class "animal". In the case of a polymorphic object, some details of its type can be selectively ignored, for example a "dog" object could be used by a function looking for an "animal". So could a "cat", because it too belongs to the class of "animal". While being accessed as an "animal", some member attributes of a "dog" or "cat" would remain unavailable, such as the "tail" attribute, because not all animals have tails.

Three properties characterize objects:

1. identity - the property of an object that distinguishes it from other objects
2. state - describes the data stored in the object
3. behavior - describes the methods in the object's interface by which the object can be used

Some terms for specialized kinds of objects include:

- Singleton object - An object that is the only instance of its class during the lifetime of the program.
- Functor (function object) - an object with a single method (in C++, this method would be the function operator, "operator()") that acts much like a function (like a C/C++ pointer to a function).
- Immutable object - an object set up with a fixed state at creation time and which does not vary afterward.
- First-class object - an object that can be used without restriction.
- Container object - an object that can contain other objects.
- Factory object - an object whose purpose is to create other objects.
- Metaobject - an object from which other objects can be created (Compare with class, which is not necessarily an object)
- Prototype - a specialized metaobject from which other objects can be created by copying
- God object is an object that knows *too much* or does *too much*. The God object is an example of an anti-pattern.

See also

- Object lifetime
- Object copy
- Design patterns
- Business object
- Actor model
- Sun Microsystems, Inc. tutorial (http://java.sun.com/docs/books/tutorial/java/concepts/)

Retrieved from "http://en.wikipedia.org/wiki/Object_%28computer_science%29"

Categories: Wikipedia articles needing context | Object-oriented programming

---

Exhibit A to April 3, 2006 Response After Final
Appln. No. 09/800,476